Clustering algorithms on GPU

Team 18

DEEPAM SARMAH

1 INTRODUCTION

In the day and age of data science, there has been an increased demand for clustering algorithms. Clustering algorithms cluster similar data points based on their characteristics or features. They are helpful in many cases. To name a few, they help segment the data into meaningful groups based on a standard metric (these can be maximum revenue, average life expectancy, etc.) and anomaly detection, where clustering algorithms can help identify outliers or anomalies in the data by identifying data points that do not belong to any cluster (this is helpful in cancer detection). GPUs help increase the efficiency of clustering algorithms mainly due to the following: Many clustering algorithms are computationally intensive, and hence the GPU could parallelize these computations. Another benefit is that GPUs provide a larger memory bandwidth and processing power, making GPUs ideal for clustering large data-sets. In this project, I aim to compare the serial and parallel implementations of K-Means, DBSCAN, and Mean-Shift clustering algorithms and compute speedup statistics on the same.

2 LITERATURE REVIEW

I reveiwed research papers and implementation of K-Means, DBSCAN and Mean-Shift algorithm.

2.1 K-Means Algorithm

The K-Means algorithm was first proposed by MacQueen in 1967 [1] and then later enhanced by Hartigan and Wong [2] in 1979. The K-Means algorithm is an unsupervised clustering algorithm in which the algorithm starts by randomly selecting a number of data points as the initial centroids for each cluster. Then, each data point in the dataset is assigned to the cluster whose centroid is closest to it. Next, the centroids are recalculated as the mean of all data points in the cluster. This process of assigning points to the closest cluster and recalculating centroids is repeated until a maximum number of iterations is reached. The result is k clusters, each represented by a centroid. There have been implementations of K-Means on GPUs using CUDA by Reza Farivar, et al. [3] in 2008. The K-Means algorithms is useful as it can handle large data-sets and can converge to the solution quickly. The most frequently used version of K-Means is the Lloyd's algorithm developed by Lloyd [4]

2.2 DBSCAN Algorithm

Density-based spatial clustering of applications with noise (DBSCAN) is a clustering algorithm proposed by Martin Ester, et al. [5] in 1996 commonly used in non-spherical datasets. A GPU implmentation of DBSCAN was proposed by Guilherme Andrade, et al. [6] in 2013. DBSCAN works by defining two important parameters: the radius ϵ and the minimum number of points required to form a cluster, $minPts$. It starts with a random data point, and builds a cluster around it by identifying all points within a distance of ϵ. If the number of points in this region is greater than or equal to $minPts$, then a new cluster is formed. Otherwise, the point is marked as noise or an outlier. DBSCAN then continues to grow the cluster by finding new points within the ϵ radius and adding them to the cluster until no new points can be added. The algorithm repeats this process for all unvisited points in the dataset, assigning them to clusters or marking them as noise. A parallel implementation has been developed by Wang, et al. [7] in 2015 which claims to speedup serial DBSCAN by 97 times. Wang's paper finds all the core points and then proceeds onto assiging a unique cluster to each core point. Post that it merges all the clusters.

Author's address: Deepam Sarmah, deepam20050@iiitd.ac.in.

2.3 Mean-Shift Algorithm

A description of the Mean-Shift algorithm is found in the paper by K Fukunaga, et al. [8] in 1975. The mean shift algorithm works by by randomly selecting a data point and defining a window around it. The mean of all data points within the window is then computed, and the window is shifted towards this mean. This process is repeated for each window until the window no longer moves or changes significantly. At the end of the process, each data point is assigned to the nearest converged window, which represents a cluster. The Mean-Shift algorithm is advantageous as it can handle large data-sets with many features and is suitable for clustering data-sets with complex cluster shapes.

3 MILESTONES

The identified milestones are:

S. No.	Milestone	Member
	Mid evaluation	
1	Implement K-Means on CPU	Deepam
2	Implement DBSCAN on CPU	Deepam
	Final evaluation	
3	Implement K-Means on GPU	Deepam
4	Implement DBSCAN on GPU	Deepam
5	Implement Mean-Shift on CPU	Deepam
6	Implement Mean-Shift on GPU	Deepam
7	Compare and analyze GPU code using profiler	Deepam

4 APPROACH

I have considered input vectors to be two dimensional for all the clustering algorithms.

4.1 K-Means

Algorithm 1: K-Means Algorithm (Lloyd's Algorithm)

```
Input: Dataset $D = p_1, p_2, ..., p_n$, Number of clusters $K$, Number of iterations $I$
Output: Clusters formed from D
Randomly select $K$ centroids $C = c_1, c_2, ..., c_K$ from $D$;
while Number of iterations is less than I do
    for $i \leftarrow 1$ to $n$ do
        Assign $p_i$ to the cluster with the closest centroid: $c_j = \arg\min_{c \in C} |p_i - c|^2$;
    end
    for $j \leftarrow 1$ to $K$ do
        Update the centroid of the $j^{th}$ cluster as the mean of all the points in that cluster:
          $c_j = \frac{1}{|S_j|} \sum_{p_i \in S_j} p_i$;
    end
end
return Clusters formed;
```

Algorithm Analysis

The K-Means algorithm which I have implemented is based on Lloyd's algorithm. The time complexity of the serial version is $O(NKI)$, where N is the number of data points, K is the number of clusters to be made, and, I is the number of iterations the algorithm runs for. The following is my analysis for 1 iteration of K-Means:

- Serial runtime $T_s = O(NK)$
- Parallel runtime $T_p = O(K)$
- Cost of Parallel runtime $pT_p = O(pK)$, p is the number of processors
- Total Overhead $T_o = pT_p - T_s = (p - N)K$
- Speedup $S = \frac{T_s}{T_p} = O(N)$
- Efficiency $E = \frac{S}{p} = O(\frac{N}{p})$
- My algorithm will be cost optimal only when $p = N$.

Approach & Challenges

For K-Means I used OpenMP library to improve the time complexity by utilizing the fork-join model to parallelize computation for chunks of indices over the cores of my CPU. Before I had written the OpenMP variant of my CPU code, I used a profiler (in CLion) to look at flame graphs of where a large chunk of my computational resource were being used up for. I discovered that the loops which were iterating over all the data points was the bottleneck. I parallelized that part using OpenMP and again analyzed flame graphs. This time for a certain test sample of 10^5 points I had gained a speedup of 10x over the non-OpenMP code. I applied various optimizations only while keeping the correctness of the algorithm in check.

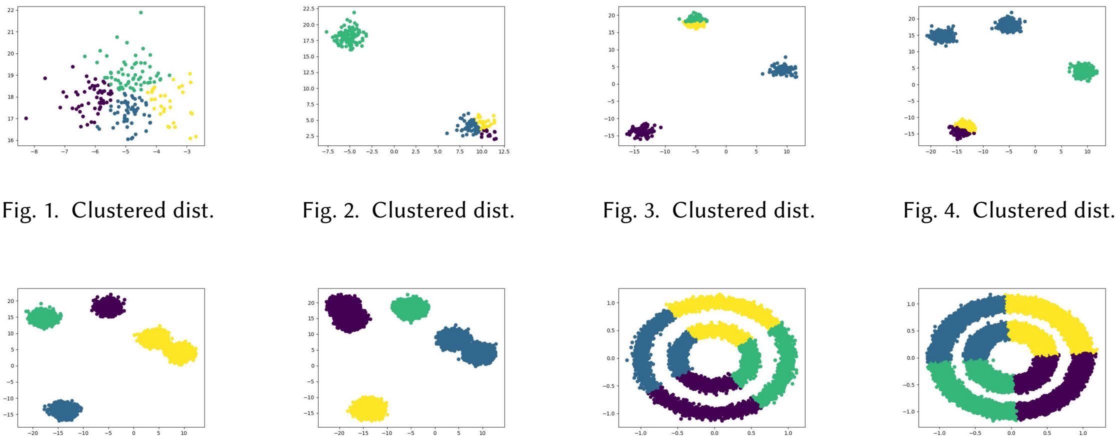

Fig. 1. Clustered dist.

Fig. 2. Clustered dist.

Fig. 3. Clustered dist.

Fig. 4. Clustered dist.

Fig. 5. Clustered dist.

Fig. 6. Clustered dist.

Fig. 7. Circular Dist.

Fig. 8. Circular Dist.

Fig. 9. K-means Clustering Plots with $K = 4$

4.2 DBSCAN

Algorithm 2: DBSCAN (CPU) Algorithm

```
Input: Dataset $D = p_1, p_2, ..., p_n$, radius $\epsilon$, minimum number of points $MinPts$
Output: Clusters formed from $D$
for $i \leftarrow 1$ to $n$ do
    if $p_i$ has not been visited then
        Mark $p_i$ as visited;
        // SpatialQuery($p_i$, $\epsilon$) returns all points at a distance of $eps$ from $x_i$
        $S \leftarrow SpatialQuery(\mathrm{p}_i, \epsilon)$
        if $|S| < minPts$ then
            Mark $p_i$ as noise;
        end
        else
            Create a new cluster $C$;
            Add $p_i$ to $C$;
            for each unvisited $p_j \in S$ do
                Mark $p_j$ as visited;
                $T \leftarrow \mathrm{SpatialQuery}(p_j, \epsilon)$
                if $|T| \geq minPts$ then
                    $S \leftarrow S \cup T$
                end
            end
        end
    end
end
return Clusters formed from $D$;
```

Algorithm Analysis

DBSCAN algorithm in its most naïve form runs in $O(N^2)$, where N is the number of data points. As N approaches a large value it is evident that this naïve version wouldn't be good for large data-sets. The optimized version of DBSCAN has an average time complexity of $O(N \log N)$. This is attained with the help of a spatial data structure. For the purposes of this project I have used R-Tree which is available in the Boost library. This way the time required for SpatialQuery(p_i, ϵ) is $O(\log N + P)$, where P is the number points at a distance of ϵ from p_i. I have used the variant of R-Trees called R*-trees as they result in better spatial query performance compared to other types of R-Trees such as linear and quadratic. The only drawback of R*-Trees is that they their construction time is larger than linear and quadratic R-Trees. To improve the construction time I have used OpenMP. The following is my analysis for DBSCAN:

- Serial runtime $T_s = O(N \log N)$
- Parallel runtime $T_p = O(N)$
- Cost of Parallel runtime $pT_p = O(pN)$, p is the number of processors
- Total Overhead $T_o = pT_p - T_s = (p - \log N)N$
- Speedup $S = \frac{T_s}{T_p} = O(\log N)$
- Efficiency $E = \frac{S}{p} = O(\frac{\log N}{p})$
- My algorithm will be cost optimal only when $p = \log N$.

```
Algorithm 3: DBSCAN GPU
Function X, $eps$, $minPts$:
    Find all core points and mark them in the $core$ array.
    Merge all core points and assign labels in $Label$ array.
    Merge border points to any valid core point at a distance of $eps$ and assign border points the core
      point's label in the $Label$ array.
    return $Label$
end
```

Approach & Challenges

The DBSCAN algorithm was the hardest clustering algorithm to parallelize. I was able to optimize the serial code using R*-Tree which is available in the Boost library but when it came to the CUDA code there wasn't a native RTree implementation provided in the CUDA Toolkit. Because of this I had to resolve to the research paper by [7].

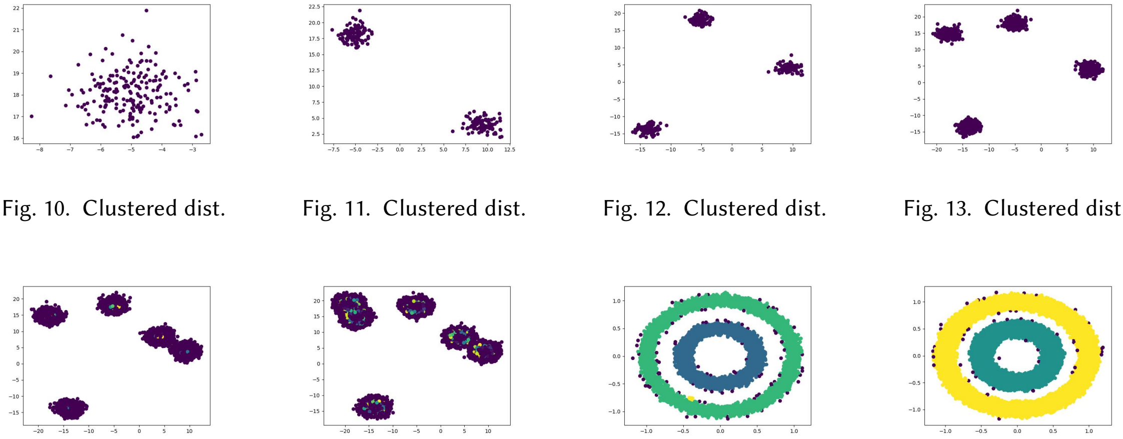

Fig. 10. Clustered dist.

Fig. 11. Clustered dist.

Fig. 12. Clustered dist.

Fig. 13. Clustered dist.

Fig. 14. Clustered dist.

Fig. 15. Clustered dist.

Fig. 16. Circular Dist.

Fig. 17. Circular Dist.

Fig. 18. DBSCAN Clustering Plots

4.3 Mean-Shift

Algorithm Analysis

Mean-Shift algorithm runs in $O(TN^2)$ where N is the number of points that are required for clustering and T is the number of iterations required for co. This is the time complexity of a serial algorithm when the Gaussian Kernel is used, $K(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{x^2}{2\sigma^2}\right)$. My analysis for 1 iteration of Mean-Shift:

- Serial runtime $T_s = O(N^2)$
- Parallel runtime $T_p = O(N)$
- Cost of Parallel runtime $pT_p = O(pN)$, p is the number of processors
- Total Overhead $T_o = pT_p - T_s = (p - N)N$
- Speedup $S = \frac{T_s}{T_p} = O(N)$

- Efficiency $E = \frac{S}{p} = O(\frac{1}{p})$
- My parallel algorithm will be cost optimal only when $p = N$.

Algorithm 4: Mean Shift Algorithm with Gaussian Kernel

```
Function MeanShift(X, h):
    Initialize $\mathbf{X}_{\text{prev}} = \mathbf{X}$;
    for iter = 1 to T do
        for $i = 1$ to N do
            Compute the mean shift vector $\mathbf{m}_i = \frac{1}{K}\sum_{j=1}^{K} \mathbf{x}_j \cdot K(\frac{\|\mathbf{x}_j - \mathbf{x}_i\|}{h})$, where $K$ is the number of
              neighbors within a bandwidth $h$ and $K(\cdot)$ is the Gaussian kernel;
        end
        Update $\mathbf{X} = \mathbf{m}$;
        Set $\mathbf{X}_{\text{prev}} = \mathbf{X}$;
    end
    return X;
end
```

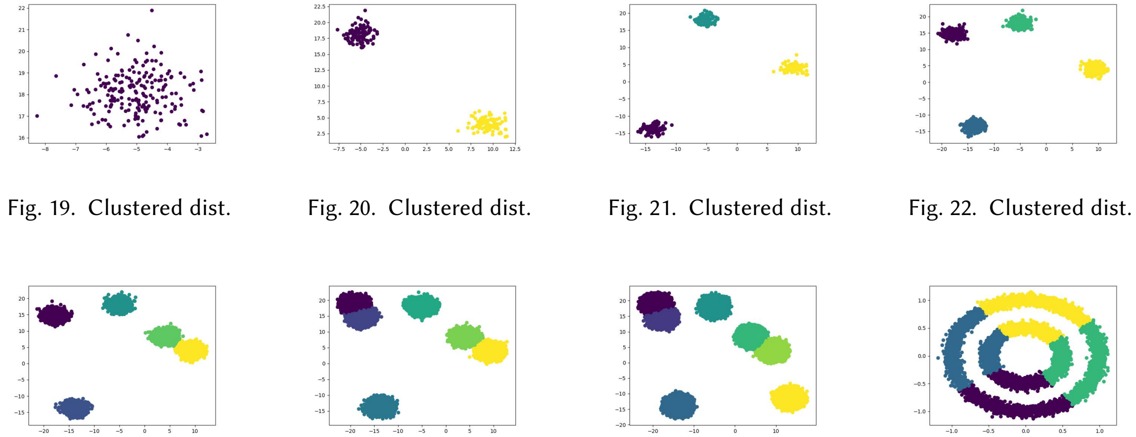

Fig. 19. Clustered dist.

Fig. 20. Clustered dist.

Fig. 21. Clustered dist.

Fig. 22. Clustered dist.

Fig. 23. Clustered dist.

Fig. 24. Clustered dist.

Fig. 25. Circular Dist.

Fig. 26. Circular Dist.

Fig. 27. Mean-Shift Clustering Plots

Approach & Challenges

The pain point in Mean-Shift was with the CPU implementation. The way Mean-Shift works with Gaussian Kernel makes it impractical for larger data-sets. Typically in the order of $10,000$. For data set of size 500000, the CPU code ends up taking 1 day. Because of the fact that GPUs have a large number of cores for performing parallel tasks, Mean-Shift clustering is suitable for GPUs. This can also be seen in the Results section.

5 RESULTS

5.1 K-Means

Table 1. Benchmarking OpenMP K-Means by taking average over 5 times, K = 4 and No. Of Iterations = 50

N	**Time (in ms)**
10	1.1987
100	0.8049
1000	4.2594
10000	32.5380
100000	321.5944
1000000	3058.8852
10000000	29443.5486

In the OpenMP implementation of Lloyd's algorithm for K-Means time complexity grows linearly with respect to the input size. We can notice a slight drop in time when N goes from 10 to 100. This time drop I suspect is due the added overhead that would be required for forking different threads and then joining them together. For small N this overhead is large. The time complexity doesn't improve for larger N mainly due to the large size of arrays and limited number of cores in my CPU.

Table 2. Performance Comparison: CPU vs. GPU

N	CPU (ms)	GPU (ms)	Speedup
10	128.724554	0.687916803	187.1222704
100	80.10202607	0.696799994	114.9569844
500	10.23848236	0.738144004	13.8705758
1000	42.37618176	0.801996803	52.8383425
5000	33.7246001	1.218681622	27.67301934
10000	34.02752513	1.645670414	20.67699877
50000	470.8680856	5.349036694	88.02857646
100000	777.338817	9.996544075	77.76075523
500000	1690.248834	37.23748474	45.39105813
1000000	3887.843754	82.8979599	46.89914877

Table 3. Performance Comparison: GPU Execution Time with Varying BLOCK_SIZE for N = 1000000

BLOCK_SIZE	GPU (ms)
32	91.26502228
64	91.5320282
128	89.16572571
256	83.18665314
512	79.70175934
1024	79.1830368

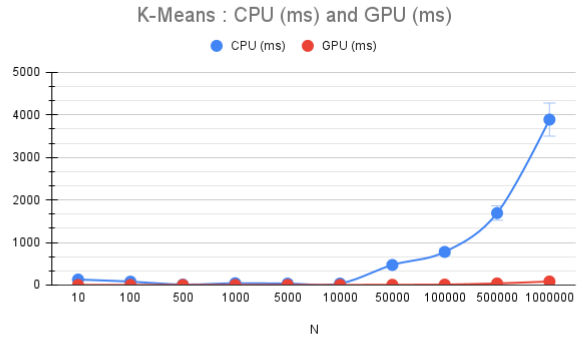

Fig. 28. K-Means CPU vs GPU

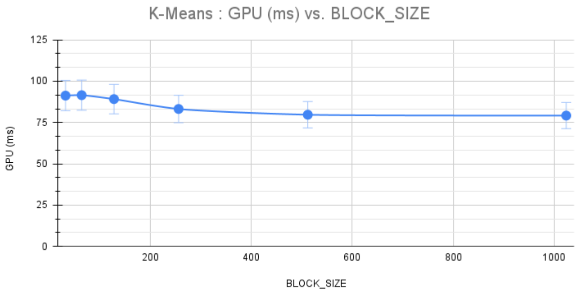

Fig. 29. K-Means GPU (ms) for N = 1000000

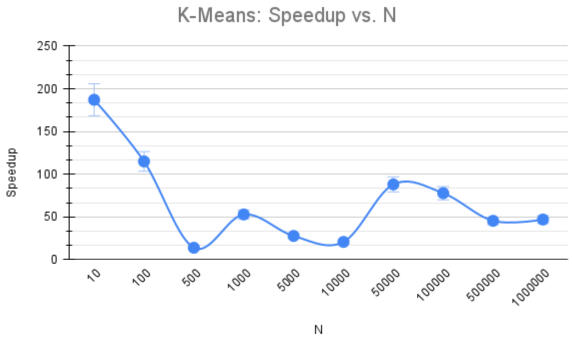

Fig. 30. K-Means Speedup vs N

5.2 DBSCAN

Table 4. Benchmarking CPU DBSCAN by taking average over 5 times, ϵ = 0.0375 and minPts = 4

N	**Time (in ms)**
10	0.1307
100	0.5741
1000	6.4857
10000	80.3703
100000	1007.9010
1000000	12201.4568
10000000	139239.5059

The $O(N \log N)$ DBSCAN works fairly well in its CPU variant. The time taken more or less increases linearly with respect to the input size which is expected. To speedup construction time of R*-trees I used OpenMP for it. But as the CPU code is predominantly serial it isn't currently possible to speedup without changing the logic of the algorithm.

Table 5. Performance Comparison: CPU vs. GPU

N	CPU (ms)	GPU (ms)	Speedup
10	0.03251452	0.032288	1.00701561
100	0.538092293	0.046566401	11.55537644
500	5.387945659	0.120172802	44.83498403
1000	10.69281772	0.229011199	46.69124377
5000	52.97995079	1.024313617	51.72239235
10000	113.332662	2.051020813	55.25670986
50000	661.0332983	25.33806725	26.08854463
100000	1328.248992	66.61844482	19.93815669
500000	7344.685016	1385.913525	5.299526184
1000000	17875.43273	5455.399707	3.276649501

Table 6. Performance Comparison: GPU Execution Time with Varying BLOCK_SIZE

BLOCK_SIZE	GPU (ms)
32	75.7190094
64	75.32182312
128	71.90326691
256	68.65078735
512	68.06060791
1024	65.09737396

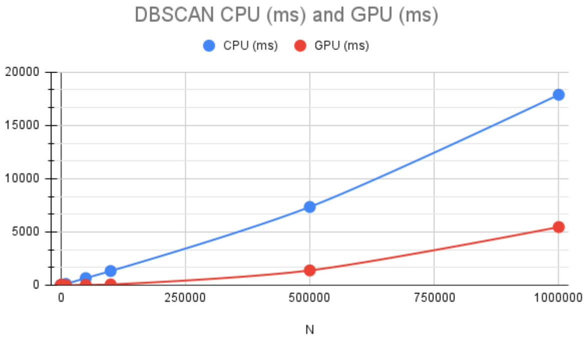

Fig. 31. DBSCAN CPU vs GPU

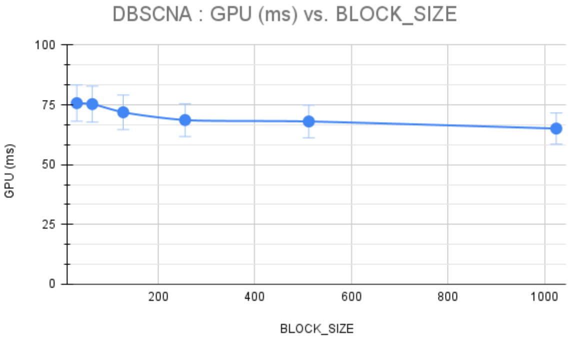

Fig. 32. DBSCAN GPU (ms) vs BLOCK_SIZE for $N = 100000$

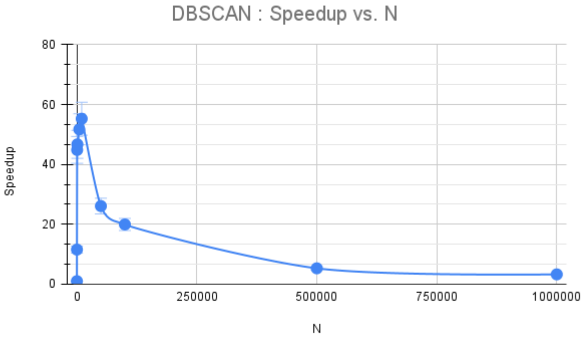

Fig. 33. DBSCAN Speedup vs N

5.3 Mean-Shift

Table 7. Performance Comparison: CPU vs. GPU

N	CPU (ms)	GPU (ms)	Speedup
10	71.61439229	0.047449601	1509.272803
100	50.3189275	0.145407999	346.053366
500	509.2631523	0.633241594	804.2162061
1000	834.4763979	1.694924831	492.3382929
5000	10687.94252	8.660582352	1234.090513
10000	36275.6454	19.73657608	1837.990807
50000	586730	240.4699921	2439.930217
100000	986730	799.0045044	1234.949233
500000	83295000	20897.67695	3985.849728

Table 8. Performance Comparison: GPU Execution Time with Varying BLOCK_SIZE

BLOCK_SIZE	GPU (ms)
32	2005.084961
64	2000.337891
128	2003.732544
256	2004.555664
512	2004.428833
1024	2006.924072

Table 9. Performance Comparison: GPU vs. GPU-Shared

N	GPU (ms)	GPU-Shared (ms)	Speedup
10	0.047449601	0.517254388	0.09173358815
100	0.145407999	0.515481591	0.2820818464
500	0.633241594	0.792985618	0.7985536933
1000	1.694924831	1.47681284	1.147691018
5000	8.660582352	7.340646362	1.179811957
10000	19.73657608	14.64811535	1.347379892
50000	240.4699921	186.786911	1.287402799
100000	799.0045044	613.0865356	1.303249147

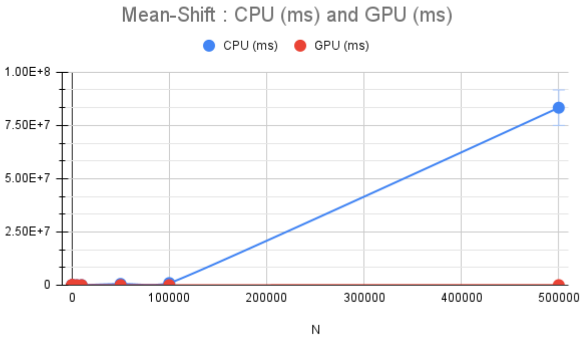

Fig. 34. Mean-Shift CPU(ms) vs GPU(ms)

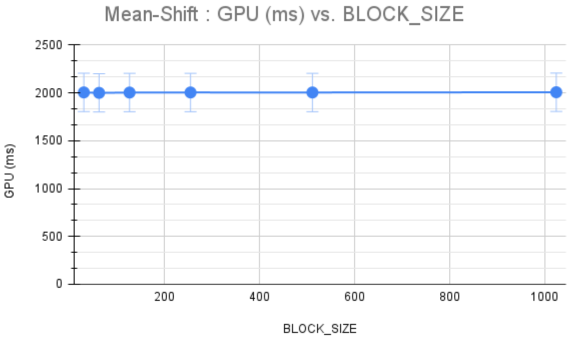

Fig. 35. Mean-Shift GPU(ms) vs BLOCK_SIZE for N = 100000

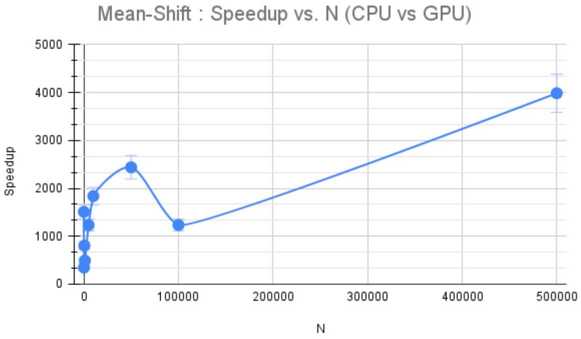

Fig. 36. Mean-Shift Speedup vs N

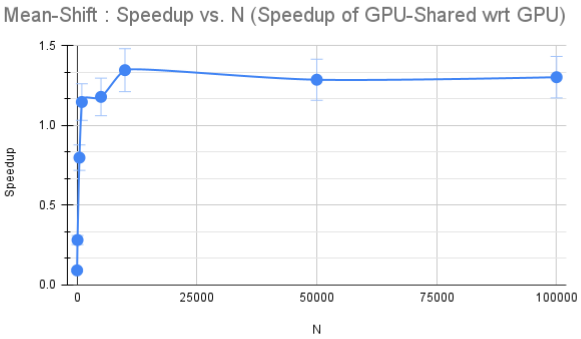

Fig. 37. Mean-Shift Speedup of GPU vs GPU-Shared

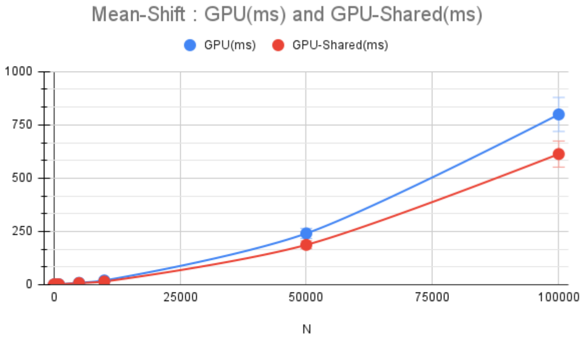

Fig. 38. Mean-Shift GPU (ms) vs GPU-Shared (ms)

6 CONCLUSION

- We can see that K-Means is the most versatile clustering algorithm in comparison to all the other ones. This is because K-Means is a form of Expectation-Maximization problem. Gaussian Mixture Model is another clustering algorithm that follows the same path where instead of K centroids we form K Gaussian distrubutions.
- DBSCAN for CUDA posed a lot many issues and didn't provide drastic speedups in the similar way K-Means provided. This is because a fully parallelized DBSCAN for CUDA requires complicated algorithms like R*-Tree, KDTrees etc which aren't standard headers included in CUDA. Despite this DBSCAN provides a good way to cluster non-hyper ellipsoid data sets which K-Means fails.
- Mean-Shift performs poorly on the CPU due to its serial runtime $O(N^2)$. On the GPU the performance is optimal.

REFERENCES

[1] J MacQueen. Classification and analysis of multivariate observations. In *5th Berkeley Symp. Math. Statist. Probability*, pages 281–297. University of California Los Angeles LA USA, 1967.

[2] John A Hartigan, Manchek A Wong, et al. A k-means clustering algorithm. *Applied statistics*, 28(1):100–108, 1979.

[3] Reza Farivar, Daniel Rebolledo, Ellick Chan, and Roy H Campbell. A parallel implementation of k-means clustering on gpus. In *Pdpta*, volume 13, pages 212–312, 2008.

[4] S. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[5] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, page 226–231. AAAI Press, 1996.

[6] Guilherme Andrade, Gabriel Ramos, Daniel Madeira, Rafael Sachetto, Renato Ferreira, and Leonardo Rocha. G-dbscan: A gpu accelerated algorithm for density-based clustering. *Procedia Computer Science*, 18:369–378, 2013.

[7] Bingchen Wang, Chenglong Zhang, Lei Song, Lianhe Zhao, Yu Dou, and Zihao Yu. Design and optimization of dbscan algorithm based on cuda, 2015.

[8] Keinosuke Fukunaga and Larry Hostetler. The estimation of the gradient of a density function, with applications in pattern recognition. *IEEE Transactions on information theory*, 21(1):32–40, 1975.